3-1-2010

Software Development Methodologies, Trends, and Implications

Xihui Zhang
xzhang6@una.edu

Tao Hu

Hua Dai

Xiang Li

SOFTWARE DEVELOPMENT METHODOLOGIES, TRENDS, AND IMPLICATIONS

Xihui Zhang
University of North Alabama
xzhang6@una.edu

Tao Hu
King College
thu@king.edu

Hua Dai
University of Wisconsin-La Crosse
dai.hua@uwlax.edu

Xiang Li
East China Normal University
xli@geo.ecnu.edu.cn

ABSTRACT

The practice of software development has evolved steadily over the decades. Numerous methods and models have been proposed to enhance its efficiency and effectiveness. This paper reviews these methods and models, identifies the latest trends in the industry, and discusses their implications.

Keywords

Software development methods and models, software testing, life cycle models, agile methods.

INTRODUCTION

As software is becoming critical to almost every organization, software development, the set of activities that produce software, has become an important topic to software development educators, students, practitioners, and researchers. The practice of software development has steadily evolved from its beginning half a century ago. Numerous methods and models (e.g., life cycle models and agile methods), have been proposed to enhance software development efficiency and effectiveness. Currently, life cycle models are the predominant models used in software development, especially in large software development organizations; however, agile methods are on the rise, gaining ground on the life cycle models. This paper reviews these software development methodologies (i.e., methods and models), identifies the latest trends in the industry, and discusses their implications. The review of methodologies, the identification of these trends, and the discussion of their implications will be useful to software development educators, students, practitioners, and researchers.

SOFTWARE DEVELOPMENT METHODS AND MODELS

This section provides a review of software development methods and models, including the analysis-coding model, life cycle models, and agile methods. For each method or model, its strengths and weaknesses are identified. Note that these methods and models are "not mutually exclusive" (Sommerville, 2007, p. 65); indeed, they are often used together, especially for the development of large, complex, integrated, and real-time systems.

Analysis-Coding Model

In the 1950's, the software development process had only two steps: an analysis step followed by a coding step, as shown in Figure 1 (Royce, 1970). There was very little emphasis on testing. As such, there were no professional testers, and testing was often carried out *implicitly* by developers or end users to ensure that the program could run and also meet the expectations. This analysis-coding model was useful for small programs, but it became practically ineffective and inefficient for larger programs (Royce, 1970).

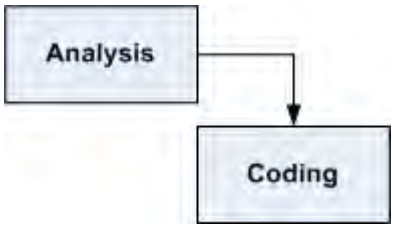

Figure 1. The Analysis-Coding Model (adapted from Royce, 1970)

Life Cycle Models

Subsequently, life cycle models were developed, with the intent to bring control and order into the software development process. Life cycle models divide the software development process into clear-cut phases, typically including steps such as analysis, design, coding, testing, and implementation (McKeen, 1983). Depending on the workflows among the phases, life cycle models can be categorized into sequential models (e.g., waterfall model, traditional V model, and PV model), progressive models (e.g., phased model), and iterative models (e.g., spiral model).

Sequential Life Cycle Models

Sequential life cycle models are those that have well-defined phases, with the development effort progressing through these phases (IPL, 1997). Examples include the waterfall model, the traditional V model, and the PV model.

Waterfall model. The waterfall model includes a set of sequential phases that flow downwards like a waterfall. These phases vary but typically include phases such as requirements analysis, program design, coding, testing, and operations (Royce, 1970). Waterfall models have well-defined boundaries and responsibilities for the stakeholders. The development process is normally well-documented because documents generated in the previous phase are required to be signed off before the development proceeds to the next phase (Sommerville, 2007). The major problem of the waterfall model is its inflexibility. It is especially ineffective and inefficient in "[responding] to changing customer requirements" (Sommerville, 2007, p. 67). With a waterfall model, testing normally gets started only after coding is complete. When defects are found, previous phases have to be revisited in order to fix them. This tends to cause a project to run over time and over budget (Tsai et al., 1997).

Traditional V model. The phases in a traditional V model are similar to those in a waterfall model. However, with the traditional V model, each testing activity is mapped to some development activity, as shown in Figure 2 (Pyhäjärvi and Rautiainen, 2004). On the left side, development activities, including requirements analysis, high-level design, low-level design, and coding, proceed from top to bottom. On the right side, testing activities, including unit testing, integration testing, system testing, and acceptance testing, are completed in a bottom-up fashion. Traditional V models have the same advantages and disadvantages as those of waterfall models.

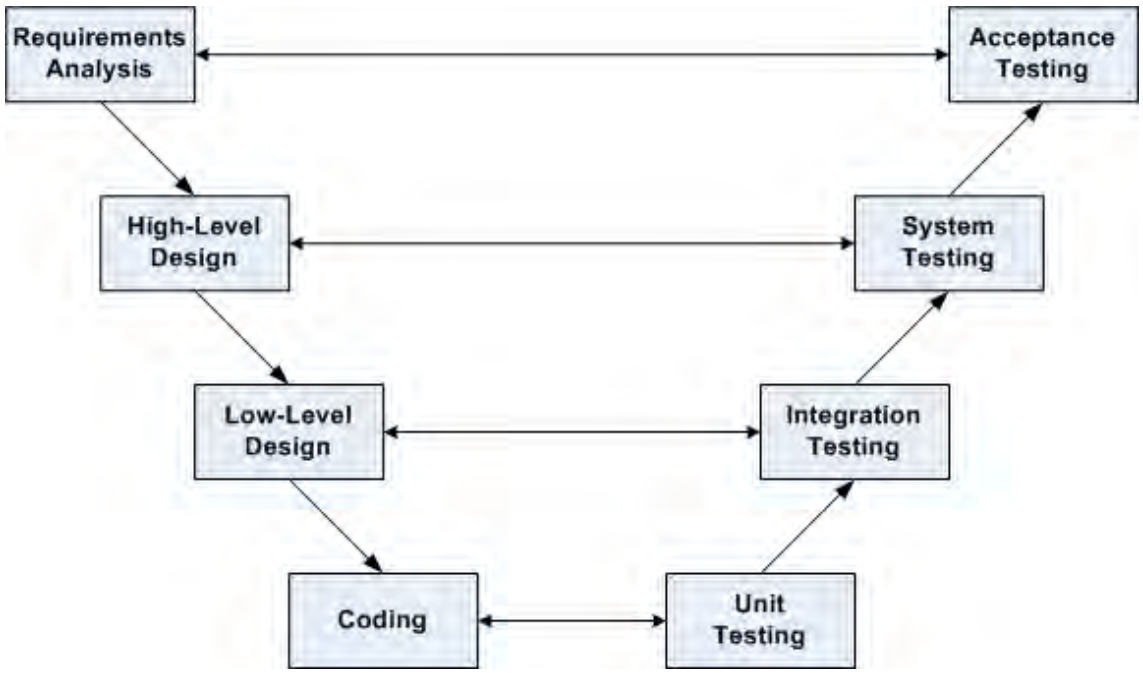

Figure 2. The Traditional V Model (adapted from Pyhäjärvi and Rautiainen, 2004)

PV model. A PV model is similar to a traditional V model. However, the PV model separates each testing activity into two parts: test planning and test execution. Test plans (e.g., test specifications) are developed along with each development

activity, and the tests will be executed in reverse order after coding is complete, as shown in Figure 3 (Sommerville, 2007). Compared to traditional V models, PV models have an additional advantage in that the test plans can be completed earlier in the process, resulting in shortened overall development and testing time.

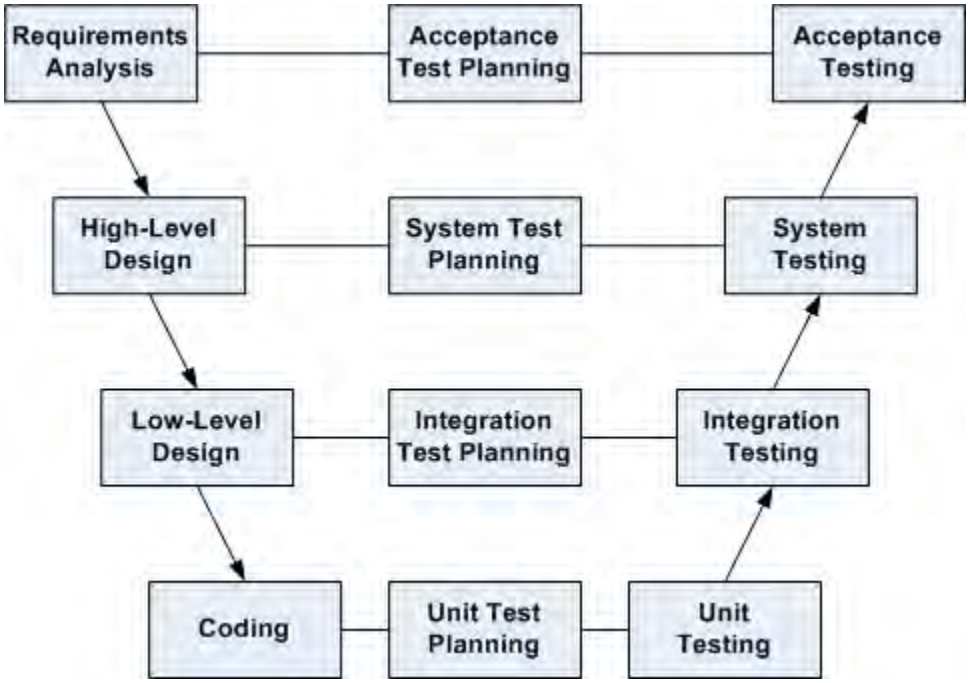

Figure 3. The PV Model (adapted from Sommerville, 2007)

Progressive Development Life Cycle Models

A progressive development life cycle model (also known as phased implementation or incremental delivery), shortens the time to market by delivering "interim" versions of the software with reduced functionality, a tradeoff that is often made in software development (IPL, 1997). Each individual phase within a progressive development life cycle may use a waterfall, traditional V, or PV model for its own development life cycle (IPL, 1997). Progressive development life cycle models provide the following major advantages (Sommerville, 2007): (1) customers can use the software after the first delivery, and therefore, refine their requirements, and (2) the highest priority functionality and features of the system are delivered first, and therefore, receive the most testing. However, with a progressive development life cycle model, it can be hard to define the "interim" versions of the software, especially when the user requirements are not specified in detail.

Iterative Life Cycle Models

A simplified iterative life cycle model consists of four major sequential phases: requirements analysis, design, implementation & test, and review (IPL, 1997). This model starts with requirements analysis, and proceeds through design, implementation & test just as the waterfall model does. However, for each cycle, a decision is made on whether the software meets all the requirements and is ready to release. If it is, the software will be tagged for final delivery; otherwise, the cycle will continue.

Spiral model. The spiral model, proposed in 1986 and refined in 1988 by Boehm, is one of the most renowned iterative life cycle models. The driving force behind the spiral model is evolutionary development, and its major goal is risk management. Each cycle of spiral involves a progression that addresses the same sequence of steps (i.e., requirements analysis, design, implementation & test), along with objective setting, risk assessment and reduction, development and validation, and planning (Boehm, 1986, 1988). With the spiral model, the highest priority features are defined and developed first, and then more features are added, refined, and implemented, incorporating feedback from end users during each cycle of the spiral. The spiral model is effective at minimizing risks which helps to decrease the project's probability of schedule and cost

overruns (Sommerville, 2007). However, the spiral model must heavily rely upon risk-assessment expertise. Furthermore, it must be highly customized to each individual software development project, limiting its re-usability across projects.

Agile Methods

Agile software development methods are basically iterative development approaches that focus on incremental specification, design, and implementation (Sommerville, 2007), while requiring full integration of testing and development (Talby et al., 2006). According to the *Manifesto for Agile Software Development* (http://agilemanifesto.org), agile methods value: (1) individuals and interactions over processes and tools, (2) working software over comprehensive documentation, (3) customer collaboration over contract negotiation, and (4) responding to change over following a plan. The intent is to produce high quality software in a cost effective and timely manner, and in the meantime, meet the changing needs of the end users.

There are many agile software development methods, with eXtreme Programming (XP) being the most prominent. Other agile methods include crystal methods, lean development, scrum, adaptive software development, dynamic systems development methods, and feature driven development (Highsmith and Cockburn, 2001; Sommerville, 2007). After a short planning stage, XP goes through analysis, design, and implementation stages quickly, as shown in Figure 4 (Dennis et al., 2005). A timebox, usually spanning one to four weeks, is used to ensure that new, enhanced software is ready to be delivered at the end of each iteration. XP principles and practices include incremental planning, small releases, simple design, test-first development, refactoring, pair programming, collective ownership, continuous integration, sustainable pace, and the presence of an on-site customer (Beck and Andres, 2005; Sommerville, 2007).

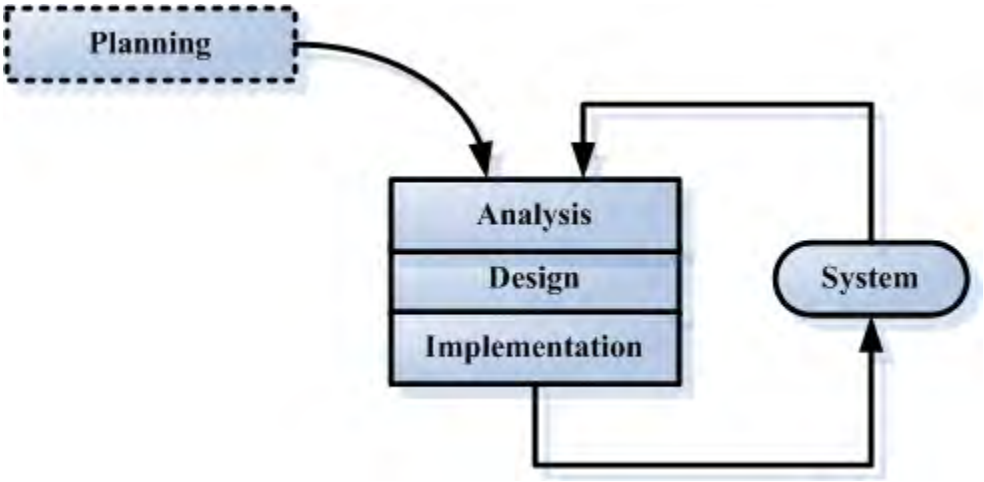

Figure 4. An XP-Based Agile Method (adapted from Dennis et al., 2005)

Test-driven development (TDD), the notion of "writing test cases that then dictate or drive the further development of a class or piece of code" (Murphy, 2005, p. 3), is an integral core practice of XP. For instance, at Parasoft Corporation (http://www.parasoft.com), at least one test should be written before coding for every task (Binstock, 2009). TDD's ability to change a class's implementation and re-test it with very little effort and time makes it a powerful tool for meeting changing client requirements during the course of a project (Murphy, 2005). Beck and Andres (2005) note that XP encourages communication, simplicity, feedback, courage, and respect. With quantitative and qualitative data, Talby et al. (2006) prove that agile software development improves development quality and productivity. For maximal success when using agile methods for software development projects, they suggest that an organization should (1) pay more attention to test design and activity execution, (2) work with professional testers, (3) plan for quality activities, and (4) manage defects.

TRENDS

Drawing upon previous reviews of software development methods and models, we identify three major trends in the software development practice. First, agile software development methods are gaining ground on life cycle models. Compared to life cycle models, agile methods offer several advantages. Agile methods can deliver working software faster. Agile methods are better at dealing with changing user requirements. And finally, agile methods promote better working relationships among all

stakeholders, including business analysts, developers, testers, end users, and project managers. Martens and Gat (2009) express it in the following way: "Agile is a systemic change. It drives cost down, quality up and service levels higher by making the entire process leaner, the entire staff more responsible, and the customer more involved" (p. 27). Moreover, empirical studies indicate that: (1) the traditional life cycle models are too inefficient and ineffective for the development of large, complex systems (Robey et al., 2001), and (2) more recent software development methodologies, such as agile methods and prototyping, can significantly improve development quality and productivity (Talby et al., 2006). Indeed, many large software development organizations, who generally use established life cycle models, are experimenting more and more with agile methods (Crispin and Gregory 2009; Lee, 2008).

Second, software testing is becoming an integral activity in the software development process. Software testing is indispensible in ensuring software quality (Cohen et al., 2004). Traditionally, testing has been viewed as a separate and distinct stage at the end of the software development process. However, the evolution of software development methodologies indicates that testing is no longer the culminating activity of the development process. Pyhäjärvi and Rautiainen (2004) argue that testing is "an integral activity in software development" and they recommend that "testing should be included early in software development" (p. 33). Schach (1996) also suggests that "testing should be performed throughout the software life cycle" (p. 277).

Third, software testers are playing a more important role in software development as a direct result of the second trend described above. The tester's role has expanded in two ways. (1) The responsibilities of testers have progressed from "error finding" (Myers, 1979) to "quality assurance" (Hetzel, 1984) to "code verification and validation" (Bentley, 2005). (2) Testers are engaged earlier and throughout the software development process, which has been proven to be beneficial to a project team's performance. For instance, Waligora and Coon (1996) present quantitative evidence that, by starting testing earlier in the development process, project performance, in terms of cost and cycle time, is improved without sacrificing the overall quality of the software. Engaging testers earlier also enables the team to catch defects earlier in the software development process. This helps to cut development costs, because fixing a defect detected in later phases tends to increase costs by one or more orders of magnitude (Tyran, 2006).

IMPLICATIONS

The trends in software development methodologies identified above have important implications, especially to the IS research community. First, as more software development organizations adopt agile methods (Trend 1), empirical studies are needed to clarify the impact of using those methods. Research questions may include: (1) Does the use of agile methods lead to a better quality product? (2) Does the use of agile methods lead to higher job satisfaction for the development team members? (3) Does the use of agile methods lead to improved working relationships among team members? (4) Does the use of agile methods lead to improved adherence to project budgets and schedules? (5) Does the use of agile methods lead to a higher level of project success? A better understanding of these questions and their answers would help any software development organization make critical decisions about adopting agile methods.

Second, as software testing is becoming an integral activity in software development (Trend 2), empirical studies are needed to explore the following research questions: (1) What are the best approaches to integrating testing into development? (2) How much, how often, and how intense should testing be done? (3) What will be the impact of adopting these approaches on overall project success?

Third, as software testers are playing a more important role in the software development process (Trend 3), empirical studies are needed to investigate the following: (1) How does the more important role that testers are playing in the software development process impact the working life of the testers? (2) How does this change impact the working relationship between testers and developers? (3) What additional skills are needed for these testers to be better prepared for this newly expanded role?

CONCLUSION

In this paper, we have reviewed software development methodologies, identified the latest trends of the practice, and discussed their implications. The contribution of this paper is threefold. The review of software development methodologies can help software development educators and students gain a solid understanding of the subject. The identification of the latest trends within the practice can help software development practitioners make better strategic and tactical development and career decisions. The implications discussed can serve to guide interested software development researchers in future research. It is our hope that this paper will instill knowledge, inspire creativity, and generate actions that are related to improved software development.

REFERENCES

1. Beck, K. and Andres C. (2005) *Extreme programming explained: Embrace change,* 2nd edition. Boston, MA: Addison-Wesley.
2. Bentley, J. E. (2005) Software testing fundamentals – Concepts, roles, and terminology. *Proceedings of SAS Conference,* April 10-13, 2005, Philadelphia, Pennsylvania.
3. Binstock, A. (2009) Seriously pragmatic agility. *Software Development Times,* December 15, 236, 29.
4. Boehm, B. W. (1986) A spiral model of software development and enhancement. *Software Engineering Notes,* 11, 4, 23-34.
5. Boehm, B. W. (1988) A spiral model of software development and enhancement. *Computer,* 21, 5, 61-72.
6. Cohen, C. F., Birkin, S. J., Garfield, M. J., and Webb, H. W. (2004) Management conflict in software testing. *Communications of the ACM,* 47, 1, 76-81.
7. Crispin, L. and Gregory, J. (2009) *Agile testing: A practical guide for testers and agile teams*. Boston, Massachusetts: Addison-Wesley.
8. Dennis, A., Wixom, B. H., and Roth, R. M. (2005) *Systems analysis and design,* 3rd edition. New York, NY: Wiley.
9. Hetzel, W. C. (1983) *The complete guide to software testing.* Wellesley, MA: QED Information Sciences.
10. Highsmith, J. and Cockburn, A. (2001) Agile software development: The business of innovation. *IEEE Computer,* 34, 9, 120-122.
11. IPL (1997) Software testing and software development lifecycles. Information Processing Ltd.
12. Lee, E. C. (2008) Forming to performing: Transitioning large-scale project into agile. *Proceedings of AGILE 2008* (pp. 106-111), Toronto, Ontario, Canada.
13. Martens, R. and Gat, I. (2009) Wrestling with scaling software agility. *Software Development Times*, December 15, 236, 27.
14. McKeen, J. D. (1983) Successful development strategies for business application systems. *MIS Quarterly,* 7, 3, 47-56.
15. Murphy, C. (2005) Improving application quality using test-driven development (TDD). *Methods & Tools*, 13, 1, 2-17.
16. Myers, G. J. (1979) *The art of software testing.* New York, NY: John Wiley & Sons.
17. Pyhäjärvi, M. and Rautiainen, K. (2004) Integrating testing and implementation into development. *Engineering Management Journal,* 16, 1, 33-39.
18. Robey, D., Welke, R., and Turk, D. (2001) Traditional, iterative, and component-based development: A social analysis of software development paradigms. *Information Technology and Management*, 2, 1, 53-70.
19. Royce, W. W. (1970) Managing the development of large software systems. *Proceedings of IEEE WESCON (WESCON 1970),* August 1970, 1-9.
20. Schach, S. R. (1996) Testing: Principles and practice. *ACM Computing Survey*, 28, 1, 277-279.
21. Sommerville, I. (2007) *Software engineering,* 8th edition. Boston, MA: Addison-Wesley.
22. Talby, D., Keren, A., Hazzan, O., and Dubinsky, Y. (2006) Agile software testing in a large-scale project. *IEEE Software,* 23, 4, 30-37.
23. Tsai, B.-Y., Stobart, S., Parrington, N., and Thompson, B. (1997) Iterative design and testing within the software development life cycle. *Software Quality Journal,* 6, 4, 295-309.
24. Tyran, C. K. (2006) A software inspection exercise for the systems analysis and design course. *Journal of Information Systems Education*, 17, 3, 341-351.
25. Waligora, S. and Coon, R. (1996) Improving the software testing process in NASA's Software Engineering Laboratory: NASA's Software Engineering Laboratory.